JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2000; **12**:79–102

Research

Stability assessment of evolving industrial object-oriented frameworks

Michael Mattsson*,† and Jan Bosch

*Department of Software Engineering and Computer Science,
University of Karlskrona/Ronneby, S–372 25 Ronneby, Sweden*

SUMMARY

Object-oriented framework technology has become a common reuse technology in software development. As with all software, frameworks evolve over time. Once the framework has been deployed, new versions of a framework potentially cause a high maintenance cost for the products built with the framework. This fact, in combination with the high costs of developing and evolving a framework, make it important for organizations to achieve a controlled and predictable evolution of the framework's functionality and costs. We present a metrics-based framework stability assessment method, which has been applied on two industrial frameworks from the telecommunication and graphical user interface domains. First, we discuss the framework concept and the frameworks studied. Then, the stability assessment method is presented including the metrics used. The results from applying the method, as well as an analysis of each of the frameworks, are described. We continue with a set of observations regarding the method, including framework differences that seem to be invariant with respect to the method. A set of framework stability indicators based on the results is then presented. Finally, we assess the method against issues related to the management and evolution of frameworks, framework deployment, change impact analysis and benchmarking. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented framework; framework evolution; framework assessment; framework stability; software architecture; object-oriented metrics

1. INTRODUCTION

During recent years, object-oriented framework technology has become common technology in object-oriented software development [1–4]. It bears the promise of reduced development effort through large-scale reuse and increased quality [5]. An object-oriented application framework is a reusable asset that constitutes the basis for a number of applications in the domain captured by the framework. Examples

*Correspondence to: Michael Mattsson, University of Karlskrona/Ronneby, S–372 25 Ronneby, Sweden.
†E-mail: Michael.Mattsson@ipd.hk-r.se

Contract/grant sponsor: The Foundation for Knowledge and Competence Development; contract/grant number: 1505/97

Received 6 September 1999
Revised 21 January 2000

of application frameworks are Java AWT [6], Microsoft Foundation Classes [7] and ET++ [8] in the domain of graphical user interfaces. Examples of application frameworks in other domains include fire alarm systems [3] and measurement systems [9].

As with all software, frameworks normally evolve through a number of iterations, leading to new versions, in response to the incorporation of new or changed requirements, better domain understanding, experiences and fault corrections. Once the framework has been deployed, new versions of a framework cause potentially high maintenance cost for the products built based on the framework. This fact, in combination with the high costs of developing and evolving an application framework, indicates the importance of understanding and characterizing framework stability and evolution.

One of the main objectives for management of software development organizations is to achieve controlled and predictable evolution of the framework's functionality and its development, maintenance and framework instantiation costs. This requires that there must exist methods assisting and providing management with information about the framework and its evolution. Areas of importance in the management and evolution of an object-oriented framework are as follows.

- **Framework deployment.** Assessment of the framework to decide when it can be released for regular product development in the organization or placed on the commercial market. The assessment is necessary to avoid shipping frameworks that do not fulfil their functional and quality requirements and, consequently, will cause unnecessary maintenance and release costs. The assessment data can in some cases, if publicly available, be used by potential customers for evaluating and selecting a framework from several alternatives. For instance, the assessment may cover aspects such as reliability and robustness for its users.
- **Change impact analysis.** Another type of framework analysis is concerned with assessing the impact of changes on the next framework version, and assessing the impact on the applications built which have to replace the old framework version with a new one. The assessment results can be used by management to predict the required maintenance effort for both the framework and the applications incorporating instantiations of the frameworks.
- **Benchmarking.** Empirical information collected from the assessment of successful frameworks can be valuable in guiding the development and evolution of new frameworks and can be used to develop criteria for framework classification, e.g., benchmarking, in terms of structural stability or other framework maturity aspects [10]. This is desirable since it is difficult and expensive to develop and maintain object-oriented frameworks and empirical data are a valuable input to the cost and effort estimation activities. Currently, little information exists about the cost and benefit of the development, maintenance and use of framework technology. Empirical information about the distribution of effort in framework development, evolution and instantiation will help in the process of allocating specialists and other personnel to the phases, as well as in effort estimations. For example, Mattsson [11] presented the effort distribution for the initial development, maintenance and instantiation of a six-year industrial object-oriented framework.

In this paper we present a method, *stability assessment*, which is used for characterizing framework evolution, and consequently, addresses the issues just described.

The stability assessment method is a metrics-based assessment method for assessing the structural and behavioural stability of an evolving object-oriented framework. The notions of structural and behavioural stability are viewed as a dual property of evolvability. Through the collection of a set of architectural and class level metrics for two or more framework versions, together with the calculation

of some aggregated metrics, the obtained metric values are used for the assessment of the framework. Thus, the purpose of the method is to decide, by using the metrics defined, how structurally and behaviourally stable a framework is under evolution. Based on the experiences of applying the method on two frameworks, we present a set of framework stability indicators.

The stability assessment method has been applied to two object-oriented frameworks: one proprietary application framework in the telecommunication domain, the Billing GateWay (BGW) framework, and one commercial application framework in the graphical user interface domain, the Microsoft Foundations Classes (MFC) framework [7]. Four consecutive versions of the BGW framework, developed over a period of six years, and five versions of the MFC framework have been analysed. Mattsson and Bosch [12] have presented some of the results of applying the method. Through the application of the method on these two frameworks it is possible to evaluate the frameworks better with respect to the aforementioned issues concerning controlled and predictable framework evolution, i.e., framework deployment, change impact analysis and benchmarking. In addition, the following aspects will be discussed: the difficulty in applying the method, the need for tool support, and the experienced advantages and disadvantages of the method.

It is important to notice that the aforementioned topics and the stability assessment method are not directly intended to be used by a project manager in developing the first version of an object-oriented framework. Neither is the method fully useable for teams reusing the framework. The usefulness is for the teams maintaining and evolving the framework. In particular, it is worth noticing that a developed framework that is intended to be reused several times has to be handled by the organization as a product. Thus, a long term commitment to the maintenance and evolution of the framework is required. It is in this organizational and long-term context that the stability assessment method and the requisite data collection are justified. Through routinely and systematically applying the method and collecting the data, the organization will achieve knowledge of the evolution of object-oriented frameworks. Currently, little knowledge about the stability and evolution of object-oriented frameworks is available.

The contribution of this paper is two-fold.

- Extension and application of a framework stability assessment method on two industrial object-oriented frameworks.
- Presentation of framework stability characterizing data together with five framework stability indicators.

The organization of this paper is as follows. Section 2 presents the notion of, and terminology related to, object-oriented frameworks. The two studied frameworks are described in Section 3. The stability assessment method is described in Section 4. Section 5 contains the results from applying the method to the frameworks together with an analysis of the results. Section 6 discusses a set of observations related to the results, the method and the studied frameworks. In Section 7 the method is assessed with respect to the aforementioned management issues and related work is presented in Section 8. Section 9 concludes the paper.

2. OBJECT-ORIENTED FRAMEWORKS

In this section we present the notion of an object-oriented framework, an overview of major framework development activities, and some of the different maturity stages a framework can have.

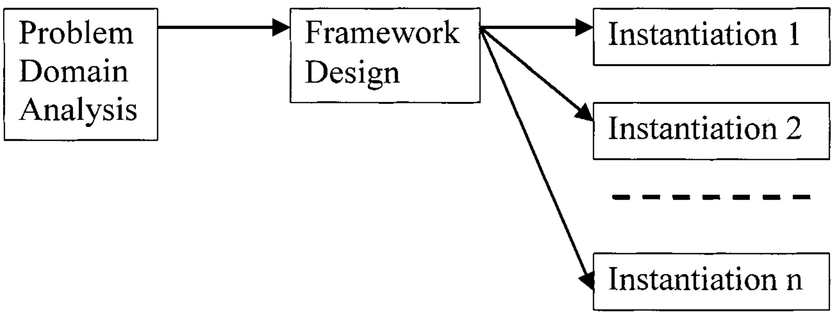

Figure 1. Development and usage of frameworks.

Object-oriented frameworks are reusable designs of all or part of a software system. A framework is described by a set of abstract classes and the way instances (objects) of those classes collaborate [4]. The development cost of frameworks is high and frameworks must be easy to use. They must provide enough features to be used and hooks for supporting features that are likely to change.

The process of developing and using an object-oriented framework consists of three major activities: problem domain analysis, framework design, and instantiation of the framework (see Figure 1).

Problem domain analysis. Domain analysis can be viewed as a broader and more extensive analysis that tries to capture the requirements of the complete problem domain including future requirements. Depending on the problem domain, different kinds of information sources are used to obtain knowledge about the domain. Examples of sources are previous experiences with the domain, domain experts and existing standards. If the problem domain is large or covers a complete application domain, specialized domain analysis methods may be used that provide systematic method steps for performing the analysis [13–15].

Framework design. The objective of this activity is to end up with a flexible framework. The activity is effort consuming since it is difficult to find the right abstractions and identify the stable and variable parts of the framework. The consequence is often a number of design iterations. To improve the extensibility and flexibility of the framework to meet the future instantiation needs, design patterns may be used [16]. The resulting framework can be a white-box, black-box or visual builder framework (see below) depending on the domain, effort available and intended users of the framework.

Instantiation of the framework. The instantiation of a framework differs depending on the kind of framework: white-box, black-box or visual builder (see below). A framework can, if small and specialized, be instantiated one or many times within the same application, or instantiated in different applications. If large and monolithic, the instantiation activity can result in an application.

Roberts and Johnson [4] describe the typical maturation path an object-oriented framework takes. We describe the four major different kinds of framework that may be developed.

White-box framework. A white-box framework relies on the inheritance mechanism as the primary instantiation technique. The instantiations of a concept are easily made through creating a subclass of

an appropriate abstract class. A disadvantage of white-box frameworks is that one has to know about internal class details and that the creation of new subclasses requires programming.

Black-box framework. A black-box framework differs from a white-box framework in that the primary instantiation technique is composition. Thus, a black-box framework allows one to combine the objects into applications and one does not need to know internal details of the objects. The use of composition to create applications implies that programming is avoided and it is possible to allow the compositions to vary at run time. Often the initial design of a framework is a white-box framework, whereas subsequent versions evolve into a black-box framework.

Visual builder framework. To evolve to the visual builder stage the framework must be a black-box framework. A black-box framework makes it possible to develop an application by connecting objects of existing classes. An application consists of different parts. First, a script is used that connects the objects of the framework and then activates them [4]. The other part is the behaviour of the individual objects, which is provided by the black-box framework. Thus, framework instantiation in the black-box case is mainly a script that is similar for all applications based on the framework. One now develops a separate graphical program encompassing the black-box framework that makes it possible to specify graphically the objects to be included in the application and their interconnections. Thus, we have a visual builder framework. The visual builder framework generates the code for the application from the specification. With the addition of the graphical features, the visual builder framework provides a user-friendly graphical interface, which makes it possible for domain experts to develop the applications by manipulating images on the screen.

Language tools. The visual builder framework creates complex composite objects. These compositions have to be easily inspected and debugged. The visual builder framework is actually a graphical domain-specific programming language. Thus, one can create specific language tools for the framework that support the inspection and the debugging of the relevant complex composite objects.

3. TWO FRAMEWORK CASES

3.1. Structural decomposition and instantiation

In this section we present the two frameworks studied. The context of the BGW framework, together with a structural view of its decomposition, is presented. In addition, we present how to instantiate the framework through a graphical configuration window, together with the major requirements causing the evolution for each framework version. We describe the MFC framework rather briefly since it is relatively well known. A more detailed presentation can be found in [7].

3.2. The Billing GateWay (BGW) framework

The Billing GateWay (BGW) framework is a major part of the Billing GateWay product developed by Ericsson Software Technology AB, and provides functionality for billing data mediation between network elements, i.e., switches, billing systems, or other post-processing systems for mobile telecommunication. The framework collects call information from mobile switches (NEs, network elements), processes the call information and mediates it to billing processing systems. Figure 2

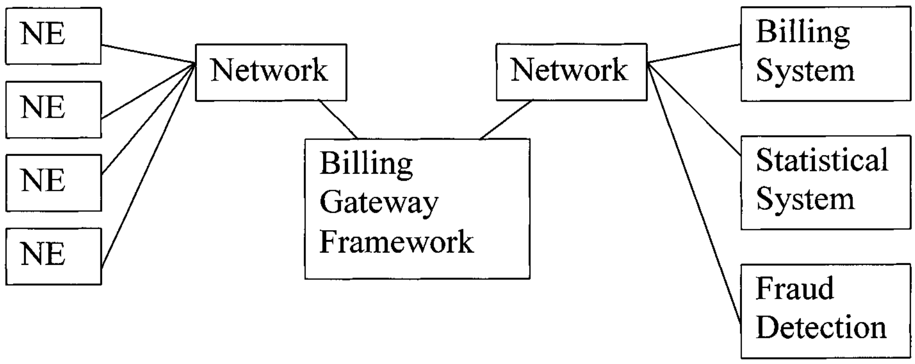

Figure 2. The Billing GateWay framework context.

presents the context of the BGW framework graphically. The driving quality requirements in this domain are reliability, availability, portability, call records throughput and maintainability.

The BGW framework is a large monolithic framework, which is intended to be instantiated into applications. The BGW framework has from the first version been a visual builder framework—i.e., a black-box framework with an associated graphical instantiation interface and application generation functionality. The BGW framework provides a number of pre-defined objects for graphical configuration of the actual application. Successively, from the second version of the BGW framework, a few specific language tools have been developed. Four versions of the framework have been developed, all in C++.

In Figure 3 a more detailed structural view of the BGW framework can be found. The BGW framework is divided into four major blocks and a set of system-wide software units. The GUI block is responsible for the user interface, and the Kernel block has responsibility for the main control, steering the flow of data from the Communication block to the Processing block and back to the Communication block. The Processing block processes the Call Data Records through different kinds of operations such as decoding, encoding, filtering, formatting, splitting and routing. The Communication block provides the interfaces to the communication protocols (protocols, e.g. RPC, and formats, e.g. FTAM, for distribution and collection). The set of system-wide software units are accessible from the other blocks and handle issues such as exception handling, threads, databases and some other system-wide issues.

Figure 4 shows a snapshot of the graphical configuration window that is used for instantiating the BGW framework. At the left in the window is a palette, where it is possible to select the different types of entities and associated operations to be performed which should constitute the instantiated application. For example, it possible to select NEs (e.g. mobile switches) and PPSs (e.g. post-processing systems), and the different kinds of processing such as filtering and formatting. At the right in the window, we see from the left side, two NEs, then two different filters, two formatters, and at the right, two post-processing systems. Each of these entities has during the configuration/instantiation to be specified in more detail, through entity-specific menus.

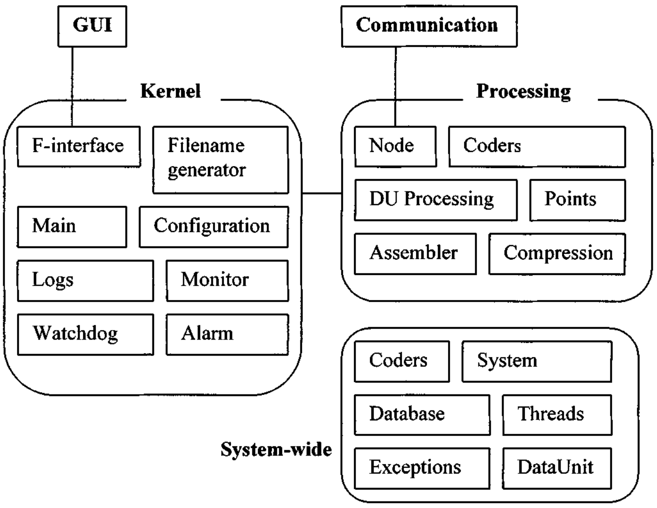

Figure 3. Structural view of the BGW framework.

When all entities have been specified, we have specified the application to be developed based on the framework, and it is possible to generate the application. All the specification data are divided into appropriate classes and objects and then the application is generated.

To give an understanding of the evolution of the BGW framework, we describe the major requirements for the four framework versions since they have impact on the structure and functionality of the framework.

Version 1. The major requirement for the development of version 1 of the framework was to develop an architecture that supported the identified variable and stable parts of the domain. The organization could here rely on the experience from a previously developed application in the same domain.

Version 2. The major requirements for version 2 of the framework were increased flexibility and robustness. This was achieved through refinement of some of the design concepts, improved encapsulation, and identification of minor domain specific libraries which provide generalized and abstract routines for data handling and formatting.

Version 3. The major requirement for the development of version 3 was the introduction of *hot billing*. Hot billing makes it possible to collect call data records from network elements and distribute them to post processing systems within seconds of call completion. Thus, hot billing opens a way for invoicing services tailored to specific customer needs, such as real-time billing and phone rental. This requirement caused major changes to the data processing in the system.

Version 4. For version 4 of the BGW framework, major requirements were portability to new operating platforms and advanced data processing such as call data record matching and rating. Matching makes it simple to collect data from different nodes and times, and thereby simplify the charging for services with more than one call data record such as distance-related charging.

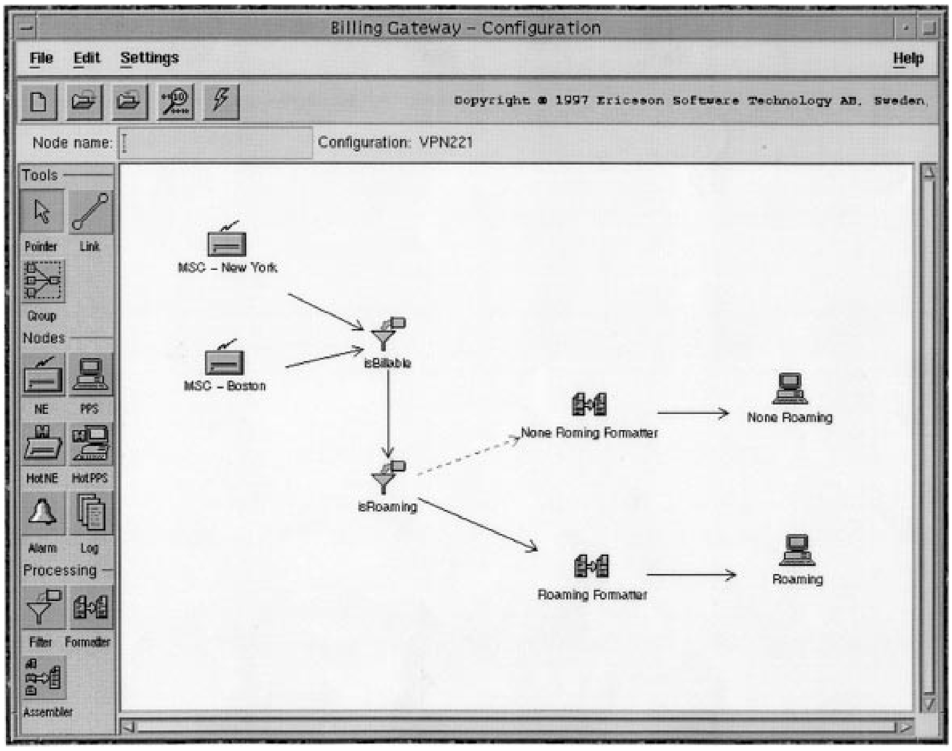

Figure 4. The graphical configuration window for the BGW framework.

Table I. Normalized BGW framework maintenance effort data.

	V1	V2	V3	V4
Normalized effort	1.000	1.266	1.761	1.620

In addition, we provided normalized effort data for the four BGW versions to indicate the magnitude of maintenance effort needed, as summarized in Table I. The effort for the development of the BGW framework versions is in the range of 10 000 to 20 000 person-hours (the company asked us not to publish the exact figures). The normalization was made with respect to the number of hours in version 1. That means, in a hypothetical example, if the development of version 1 took 10 000 hours, the development of version 2 took 12 660 hours. The duration of the development and evolution of the BGW framework was approximately six calendar years.

3.3. The Microsoft Foundation Class framework

The MFC framework addresses the domain of graphical user interfaces for Windows applications. The MFC framework has evolved through a number of versions and we have assessed the first five versions of the framework. Version 1.0 of the framework was available on the market in 1992 and version 5 was released early in 1997. Thus, the duration of the development and maintenance of the MFC framework is similar to the BGW case: around six years.

Through using the MFC framework, the developer seldom has to access the Windows API (Application Program Interface) directly. Instead, one can use the MFC classes, which provide services that wrap the functionality of the API. The framework comprises support for, among other matters:

- window handling;
- interfaces for graphical devices;
- file handling;
- exception handling;
- database handling.

A more detailed description of the MFC framework can be found in [7].

4. THE METHOD

4.1. Four steps

The stability assessment method is a metric-based approach for assessing the stability of an evolving framework. Recently, a method for quantitatively assessing stability of an object-oriented framework was proposed by Bansiya [10]. In this study we have extended the method with an additional aggregated metric, the relative-extent-of-change metric. We have applied our method on a number of versions of two industrial object-oriented frameworks, four consecutive versions of the BGW framework, and five consecutive versions of the MFC framework. The framework stability assessment method consists of the following four steps:

(i) identification of evolution characteristics;
(ii) selecting metrics to assess each evolution category;
(iii) data collection; and
(iv) analysing the changed characteristics and computing the aggregated metrics.

Based on the results of applying our method, we propose in Section 6 a set of (preliminary) framework stability indicators.

4.2. Identification of evolution characteristics

When assessing framework stability, one has to select suitable indicators. The stability assessment method focuses on measuring changes of object-oriented design components, i.e., the classes and two categories of evolution characteristics, *architectural* and *individual class*, are identified [10].

Table II. Architectural metrics.

Architectural metrics (structure)	Architectural metrics (interaction)
DSC, design size in classes in all classes	ACIS, average number of public methods
NOH, number of hierarchies classes from which a class inherits information	ANA, average number of distinct (parent) classes from which a class inherits information
NSI, number of single inheritance instances	ADCC, average number of distinct classes that a class may collaborate with
NMI, number of multiple inheritance instances	
ADI, average depth of the class inheritance structure	
AWI, average width of the class inheritance structure	

- First, the changes of architectural characteristics that are related to the *interaction* (collaborations) among classes and the *structuring* of classes in inheritance hierarchies.
- Second, the characteristics of individual classes that are related to the assessment and change of the *structure*, *functionality* and *collaboration* relationships of individual classes between versions.

The structure of objects is described and detailed by the data declarations in the class declarations. The functional characteristics are related to the object's methods. The parameter and data declarations that use user-defined objects define the collaboration characteristics of a class.

4.3. Selecting metrics to assess each evolution category

Several authors have proposed and studied metrics for object-oriented programming [17–19] and, to some extent, for object-oriented frameworks [20]. In Table II, the architectural framework structure and interaction metrics used in the method are described. The individual class metrics comprise a total of 11 metrics divided into three separate suits, functional, structural and relational, and are described in Table III. The metrics used are a subset of object-oriented metrics presented by Bansiya [17] and Chidamber and Kemerer [18].

4.4. Data collection

The metrics data were collected using the QMOOD++ tool [17]. The data were collected for each version of the two frameworks with the same tool, thereby reducing the probability of different interpretations of the metric definitions. The data for the MFC framework were obtained from [10].

Table III. Class metrics.

Structural metrics	Functional metrics	Relational metrics
NOD, the number of attributes.	NOM, count of all the methods defined.	NOA, the number of distinct classes (parents) which a class inherits.
NAD, the number of user defined objects used as attributes in a class.	NOP, the number of polymorphic methods in a class.	NOC, the number of immediate children (sub classes) of a class.
NRA, the number of pointers and references used as attributes.	NPT, the number of parameter types used in the methods of a class.	DCC, count of the number of classes that a class is directly related to. Includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CSB, the size of the objects in bytes from the class declaration. Size is computed by summing the size of all attributes declared (class size in bytes).	NPM, average of the number of parameters per method in a class.	

4.5. Analysing characteristics and computing metrics

The fourth step consists of analysing the changed characteristics and then computing the aggregated metrics. For each of the frameworks and their versions, the collected architectural data are analysed and the two aggregated, normalized-extent-of-change and relative-extent-of-change, metrics are calculated. These two metrics are described in detail in Section 5 since they are better explained in the context of the other metrics used.

5. RESULTS AND ANALYSIS

5.1. Two types of metric

We present in Table IV the metric values from the assessment of the two cases, together with an analysis of the obtained results. First, the architectural, normalized architectural and the associated aggregated metric values are presented and discussed for each of the two frameworks. Second, the class metric values with their associated aggregated metric values are presented and discussed for the frameworks.

Table IV. Architectural metric values.

	BGW				MFC				
Metric	1	2	3	4	1	2	3	4	5
DSC	322	445	535	598	72	92	132	206	233
NOH	30	35	40	44	1	1	1	5	6
NSI	265	371	434	480	66	84	117	174	194
NMI	5	1	1	1	0	0	0	0	0
ADI	1.97	2.24	2.09	2.14	1.68	2.11	2.45	2.33	2.30
AWI	0.84	0.84	0.81	0.80	0.92	0.91	0.89	0.85	0.83
ANA	2.00	2.25	2.09	2.14	1.68	2.11	2.45	2.33	2.30
ACIS	15.84	17.58	18.90	18.68	44.6	75.6	95	114	109
ADCC	3.61	4.07	3.99	3.98	6.03	7.59	9.02	9.33	8.94

5.2. Architectural metrics

5.2.1. BGW case

The architectural metric data collected for the four versions of the BGW framework shows that the number of classes nearly doubles from 322 in version 1 to 598 in version 4. The values for the number of hierarchies (NOH) and the number of single inheritance (NSI) metrics agree with the increasing value of the number of classes. In addition, the values of the average depth of inheritance (ADI) and average number of ancestors (ANA) are nearly the same due to the low number of multiple inheritance instances (NMI).

The average depth of inheritance (ADI) reflects the level of specialisation in the framework and is expected to increase in newer versions. The average width of inheritance (AWI) is expected to decrease in new versions since new classes are generally added at deeper levels in the inheritance hierarchy. In the BGW study, we find a minor increase in the average depth of inheritance (ADI) values and a minor decrease in the average width of inheritance (AWI) values. Although this is as expected, the differences between the versions could have been larger. One possible explanation may be that this is a property of black-box and visual frameworks compared to white-box frameworks.

Across the four versions, the average number of public methods (ACIS) is 15 to 19 methods, and the average number of directly coupled classes (ADCC) is 3.6 to 4.0 classes. These narrow ranges indicate that the averages are relatively constant throughout the versions.

5.2.2. MFC case

We see that the MFC has had an increase in the number of classes by more than three times, from 72 classes in version 1 to 233 classes in version 5. Regarding inheritance relationships, the MFC does not make use of multiple inheritance, and in version 4 the number of inheritance hierarchies was increased to five from only one in the previous versions. The values of the average depth of inheritance (ADI) and

Table V. Normalized architectural metrics values.

Metric	BGW 1	BGW 2	BGW 3	BGW 4	MFC 1	MFC 2	MFC 3	MFC 4	MFC 5
DSC	1	1.38	1.20	1.12	1	1.28	1.43	1.56	1.13
NOH	1	1.17	1.14	1.10	1	1	1	5	1.2
NSI	1	1.40	1.17	1.11	1	1.27	1.39	1.49	1.11
NMI	1	0.2	1	1	1	1	1	1	1
ADI	1	1.14	0.93	1.02	1	1.26	1.16	0.95	0.99
AWI	1	1	0.97	0.96	1	0.99	0.98	0.96	0.98
ANA	1	1.12	0.93	1.02	1	1.26	1.16	0.95	0.99
ACIS	1	1.11	1.08	0.99	1	1.70	1.26	1.20	0.96
ADCC	1	1.13	0.98	1	1	1.26	1.19	1.03	0.96
Aggregate change (V_i)	9	9.65	9.41	9.32	9	11.01	10.57	14.14	9.31
Normalized-extent-of-change	0	0.65	0.41	0.32	0	2.01	1.57	5.14	0.31

average number of ancestors (ANA) are the same due to the lack of multiple inheritance in the MFC. The average number of public methods (ACIS) has increased from 75.6 public methods up to 110 to 115 in the last two versions. The average number of directly coupled (ADCC) classes is relatively constant around 9.0, with the exception of the early version where the value was lower.

5.3. Normalized architectural metrics

5.3.1. Normalization computation

In Table V the normalized architectural metrics and the normalized-extent-of-change metric are presented. In the table the architectural metric values are normalized (made relative) with respect to the metric's values in the first version of the framework. The metric values for the first version of each framework are used as a base for normalization. The normalized metric values are computed by dividing the actual metric values of a version with the metric's value in the previous version for the actual framework. The normalized-extent-of-change metric is computed by summarizing the normalized metrics for a framework version. The reason for the normalization is that we work with metrics whose values are of different ranges in the aggregated metrics. The normalization is performed separately for each framework since they were developed by two independent organizations.

The aggregated metric has for the first version of the framework the same value as the number of architectural metrics (nine). Second, the normalized-extent-of-change metric is then computed by taking the difference of the aggregated metrics for the framework version, V_i, and the first framework version V_1. For the first version of a framework the normalized-extent-of-change metric is set to 0. For example, the normalized-extent-of-change metric for version 3 of the BGW framework is computed as 9.41 (aggregate metric values for version 3) minus 9 (aggregate metric value for version 1), which

is 0.41. Thus, the normalized-extent-of-change metric has the value 0.41 for version 3 of the BGW framework.

The normalized-extent-of-change metric is a *relative indicator* of the framework's architectural stability. A high value of the metric indicates relative instability of the framework structure and a low value of the normalized-extent-of-change metric indicates greater stability [10]. One way of viewing the normalized-extent-of-change metric is that it captures the enhancements of the framework, e.g., a high value indicates that the framework's architectural structure has changed significantly.

5.3.2. BGW case

For the BGW framework the normalized values for number of hierarchies (NOH) is decreasing from 1.17 to 1.10. This indicates that the addition of new inheritance hierarchies is slowly decreasing with newer versions of the framework. Another observation is that the average number of directly coupled classes (ADCC) is fluctuating just above and below 1.0. This indicates that the 'relationship complexity' is not increasing with newer versions, i.e. the number of relationships for a class is not increasing significantly in a new version of the framework. Regarding the normalized-extent-of-change metric, it is decreasing; starting from a low value, 0.68, which we interpret as the BGW framework, it is enhanced at a relatively modest pace.

5.3.3. MFC case

The normalized value for the number of hierarchies (NOH) metric makes a jump up to five in version 4 due to the introduction of five separate hierarchies in the framework. The normalized values for the number of multiple inheritance hierarchies (NMI) is constantly 1 since there are no occurrences of multiple inheritance in the framework. In the MFC framework case one can make the same observation as in the BGW framework regarding the average number of directly coupled classes (ADCC) metric. It is close to 1, indicating that the 'relationship complexity' is not increasing with newer versions of the framework. Regarding the normalized-extent-of-change metric, it is fluctuating up and down, indicating that major enhancements and restructuring of the framework's architectural structure have occurred. One example is the introduction of four new inheritance hierarchies in version 4.

For both frameworks we see that for the normalized average number of public methods (ACIS), as well as for the normalized average number of directly coupled classes (ADCC) metrics, the values are steadily decreasing. A possible explanation is that the framework is becoming more stable and modifications can be realized as more localized in the framework.

5.4. Class metrics

5.4.1. Computation and interpretation

The individual class metrics defined in Section 4 can be computed in two ways, i.e., including or excluding inherited properties. The first approach requires the analysing of a class and all its ancestors for the computation of the class metrics. Typically, changes made to the internal parts of parent classes in a framework ripple the effect of changes to all descendants of the parent class. Thus, in this study the class metrics values are counted, including the inherited properties.

Table VI. Changes in class characteristics.

	BGW					MFC					
Metrics	V1/2	V2/3	V3/4	Total units changed	% change	V1/2	V2/3	V3/4	V4/5	Total units changed	% change
NOC	19	31	47	97	2.4	7	10	12	12	41	1.8
NOA	47	13	10	70	1.7	18	11	5	0	34	1.5
DCC	89	80	141	310	7.5	29	46	77	23	175	7.7
NOM	148	142	198	488	11.9	68	84	126	120	398	17.5
NOP	105	93	117	315	7.7	64	81	119	12	276	12.1
NPT	77	125	187	389	9.5	68	87	124	93	372	16.3
NPM	147	156	190	493	12.0	68	84	124	117	393	17.2
NOD	78	126	144	348	8.5	49	47	83	25	204	8.9
NAD	69	51	2	202	4.9	6	10	21	5	42	1.8
NRA	76	88	58	222	5.4	14	38	66	15	133	5.8
CSB	77	125	187	389	9.5	48	48	90	26	212	9.3
Totals	932	1030	1361	3323	100.0	439	546	847	448	2280	100.0

The metric for each of the 11 class characteristics can be changed between two successive versions. If the value of a particular metric (e.g., number of methods, NOM) changes from one version to the next, this is defined as *one unit of change*. Since there are 11 characteristics that can be changed for a class, a class can contribute with between 0 to 11 units of change. The 11 metric values of each class are compared with the metric values of the class in its predecessor version to compute the total number of units-changed for the classes. The *total-extent-of-change* for a framework version is the sum of units-changed in all classes between a version and its predecessor. In Table VI, the 11 class metrics obtained for the framework versions are presented.

In Table VI, we see that the NOA (number-of-ancestor) metric changed values for 18 classes between version 1 and 2 of the MFC framework. The reason for this large change in the metric values was the restructuring of the MFC class hierarchy, which introduced several new classes in the middle of the hierarchy [7,10].

To assess the significance of the total-extent-of-change measure, we compare it with the maximum possible change. The maximum possible change represents the case where all the metrics would have changed values for all classes in a framework version. In Table VII, the values for the *relative-extent-of-change metric* are presented. The relative-extent-of-change is computed as the *total-extent-of-change* divided by the maximum possible change.

The reason for introducing the relative-extent-of-change metric is two-fold. Firstly, the relative-extent-of-change metric is calculated based on class metrics, rather than system-wide metrics, which is the case for the normalized-extent-of-change metric. Thus, the relative-extent-of-change metric gives us a possibility to *validate* the original proposed normalized-extent-of-change metric. Secondly, the normalized-extent-of-change metric addresses the framework's architectural structure, which implies

Table VII. Relative-extent-of-change metric values.

	BGW				MFC				
	V1/2	V2/3	V3/4	Total	V1/2	V2/3	V3/4	V4/5	Total
Actual units changed	932	1030	1361	3323	439	546	847	448	2280
Max. units of change	3542	4895	5885	14 322	792	1012	1452	2266	5522
Relative-extent-of-change	26.3%	21.0%	23.1%	23.2%	55.4%	54.0%	58.3%	19.8%	41.3%

that the metric is more focused on capturing the enhancements, e.g. addition of features, of the framework. This is in contrast to the relative-extent-of-change metric, which analyses the *core* of the framework, i.e., it only measures changes in the set of classes that exist in two consecutive versions. That means that the metric expresses how well the domain is captured and understood by the framework developers. The core of any framework version, n, consists of the classes that exist in two consecutive versions of the framework. A more formal definition is that the core is the set of classes that exist in both version $n - 1$ and version n of the framework. That means that the core does not comprise the classes which existed in version $n - 1$ and now do not exist in version n of the framework. Neither are the classes added to the framework version n part of the core. A consequence of this definition is that the core can grow with newer framework versions since it is defined in terms of two consecutive versions and not the first version of the framework. The set of added classes can be seen as representing a wider and better domain understanding captured by framework version n. This means that the core for framework version $n+1$ is expanded with the set of added classes in version n, except for those classes that have been removed in version n of the framework, and is reduced with those classes in the core for version n which do not exist in framework version $n + 1$. Thus, a low relative-extent-of-change value indicates good domain coverage and understanding, whereas a high relative-extent-of-change value indicates that the domain was not well understood and important abstractions have not been found.

5.4.2. BGW case

For the BGW framework, the majority of the changes are related to methods and method parameters, i.e., the functionality suit of metrics. For example, the number-of-methods (NOM) is 11.9%, the average number-of-parameter types (NPT) is 9.5%, and the average number-of-parameters per method (NPM) is 12.0%. An exception is the number-of-polymorphic-methods (NOP), which only represent 7.5% of the total changes. A possible explanation is that the BGW framework is intentionally designed as a visual builder framework, thus favouring parameterization before inheritance. Regarding the relative-extent-of-change metric, the values for the BGW framework are relatively low, between 21%

and 26%, indicating that the domain coverage is good, and that a relatively stable core in the framework persists largely unchanged.

5.4.3. MFC case

For the MFC framework the same pattern occurs. The majority of changes belong to the functional suit of metrics. For example, the number-of-methods (NOM) is 17.5%, the number-of-polymorphic-methods (NOP) is 12.1%, and the average number-of-parameters per method (NPM) metric is 17.2%. The relative-extent-of-change metric values for the MFC framework are high, over 50%, for the first four versions, indicating problems with understanding the domain and capturing useful abstractions.

6. DISCUSSION

6.1. Three aspects of the method

In this section we discuss a number of observations related to the results of applying the method, the method itself, and the differences between the two frameworks with respect to which the method seems to be invariant.

6.2. Observations on results of method application

From studying, in terms of stability and evolvability, the results of applying the method on the two frameworks, we offer a set of framework stability indicators.

Considering the values for the MFC and BGW frameworks (see Table IV), we see that the initial ADI values are increasing from values below 2.0, and are then successively increasing to values above 2.1. During the evolution of the frameworks, some versions have peak values for the ADI metric. For later versions, the ADI metric decreases to a level between 2.1 and 2.3. The AWI metric for the BGW framework is around 0.83 and decreases to 0.83 for the MFC framework. Thus, we formulate our first framework stability indicator that is fulfilled for the BGW and MFC frameworks as follows.

Framework stability indicator 1. Stable frameworks tend to have narrow and deeply inherited class hierarchy structures, characterized by high values for the average depth of inheritance (above 2.1) of classes and low values for the average width of inheritance hierarchies (below 0.85).

Considering from Table VIII the ratio between the number of subclasses and the total number of classes for a framework version, we see that the ratio is going towards a limit value of 0.8 for the BGW and MFC frameworks. Thus, we have our second framework stability indicator.

Framework stability indicator 2. A stable framework has an NSI/DSC (number of single inheritance/design size in classes) ratio just above 0.8 if multiple inheritance is seldom used in the framework. That is, the number of subclasses in a stable framework is just above 80%.

An analysis of the ADCC metric values for the BGW and MFC frameworks shows (see Table V) a peak in version 2 and then a slow decrease down to 1 or below in later versions. The fact that the ADCC values are close to 1 means that the relationship complexity is not increasing with newer versions, nor is it decreasing, i.e., the number of relationships for a class is relatively stable (nearly constant)

Table VIII. The NSI/DSC ratio (number of single inheritance/design size in classes).

	BGW				MFC				
Metric	1	2	3	4	1	2	3	4	5
NSI	265	371	434	480	66	84	117	174	194
DSC	322	445	535	598	72	92	132	206	233
NSI/DSC	0.82	0.83	0.81	0.80	0.92	0.91	0.89	0.84	0.83

throughout all versions of the framework. We formulate our third framework stability indicator as follows.

Framework stability indicator 3. The normalized ADCC (average of directly coupled classes) metric is going towards 1.0 or just below for stable frameworks.

The MFC framework shows a normalized-extent-of-change value beginning at 2.01 and then successively decreasing down to 0.31 for version 5 (see Table V). For version 4, the normalized-extent-of-change value is 5.14 which is an atypical value for the MFC framework. The reason for this is the introduction of four new hierarchies in version 4 of the MFC; otherwise the values would probably be around 1.5 or lower. In the case of the BGW framework, the normalized-extent-of-change metric decreases from 0.68 and ends with 0.35 for version 4. All versions have values less than 0.7. This results leads to our fourth framework stability indicator.

Framework stability indicator 4. The normalized-extent-of-change metric is below 0.4 for a stable framework.

The relative-extent-of-change metric is in one way similar to the normalized-extent-of-change metrics since it tends to indicate the stability of the framework. On the other hand, the relative-extent-of-change metric is on another abstraction level, since it is composed of a set of finer grained class metrics and is not directly measured from the architecture of the framework. In Table VII, the relative-extent-of-change metric values are presented. For example, version 1 of the BGW framework has 322 classes, which gives a theoretical maximum of 322×11 (number of metrics that can change for a class) $= 3542$ possible changes for the (previous) version. The actual number of changes between version 1 and version 2 of the BGW framework is 932, which represents a $100 \times (932 \div 3542) = 26.3\%$ change between version 1 and 2. Thus, the relative-extent-of-change metric is 26.3% for version 2 of the BGW framework.

The large BGW framework exhibits a relative-extent-of-change value just above 25% for the first transition, version 1 to version 2. For the successive transitions the value is below 25%. In the case of the MFC framework, it begins at a level above 55% for the first transition and continues to be on a level above 50% until version 5, when a large drop occurs down to 19.8%. This points to our fifth framework stability indicator.

Framework stability indicator 5. A stable framework exhibits a relative-extent-of-change value of less than 25%.

Table IX. Fulfilment of framework stability indicators.

BGW	SI1	SI2	SI3	SI4	SI5	MFC	SI1	SI2	SI3	SI4	SI5
V1	N	Y	N	N	N	V1	N	N	N	N	N
V2	Y	Y	N	N	N	V2	N	N	N	N	N
V3	N	Y	Y	Y[1]	Y	V3	N	N	N	N	N
V4	Y	Y	Y	Y	Y	V4	Y	Y	Y	N	N
						V5	Y	Y	Y	Y	Y

[1]The normalized-extent-of-change value was 0.41 for the version.

In Table IX, we see for the two frameworks which versions the framework stability indicators are fulfilled. The five framework stability indicators are labelled SI1–SI5 in the table. A 'Y' indicates that the stability indicator is fulfilled, and an 'N' that it is not fulfilled.

The BGW framework fulfils stability indicators 1 and 2 for version 2 and all but indicator 1 for version 3 (see Table IX). Version 4 of the BGW framework fulfils all the stability indicators. For the MFC framework, the situation is that it fulfils indicators 1, 2 and 3 for version 4. For version 5 of the MFC framework the stability indicators 4 and 5 also fulfilled.

The reason why the BGW framework achieves stability in earlier versions than the MFC framework is probably that in the BGW case the supplier and the customers of the framework had a closer relationship, the customers are relatively few and they are sophisticated users/domain experts. In the MFC case the customer is a mass-market with a large number of anonymous users of the framework.

The values used in the stability indicators may be questioned since they may be affected by organizational standards. However, our experiences from the application of the method on the two frameworks from different organizations give us confidence in the values and their interpretation.

6.3. Observations on the method

The differences between our assessment method and the approach of Bansiya [10] are as follows.

- The framework stability assessment method has been *applied* on two different frameworks, the BGW and MFC frameworks; the BGW framework especially represents a more typical study subject since it represents a more common situation in the software industry.
- An aggregated metric, the relative-extent-of-change metric based on individual class level metrics, which is on a higher abstraction level than the original normalized-extent-of-change metric, has been added. The introduction of the relative-extent-of-change metric gives the possibility of *validating* the originally proposed normalized-extent-of-change metric. In Table IX, we see that the stability indicator based on the relative-extent-of-change metric, stability indicator 4, indicates framework stability for versions 3 and 4 of the BGW framework and version 5 of the MFC framework. The architectural metric, normalized-extent-of-change, used in stability indicator 5 indicates framework stability for versions 3 and 4 of the BGW

framework and version 5 of the MFC framework. Thus, we see that the normalized-extent-of-change metric indicates stability in the same version as the relative-extent-of-change metric. One conclusion is that the normalized-extent-of-change metric is a good indicator of framework stability. The other aspect of the relative-extent-of-change metric is that it gives us an indication of the stability of the core of the framework, i.e., how well the domain is captured in the framework.

- As a result of applying the method a set of *experience-based framework stability indicators* has been added, which provide a more objective basis for deciding if a framework is stable or not.

Bansiya's approach [10] has been twice validated through application of the assessment method, and the method does not show any particular weakness. The same set of metrics used in Bansiya's approach has been used. There also exists a possibility to change the metric set in the assessment procedure, making the method more flexible, but in our study we have chosen to use the original set.

Both the BGW and MFC frameworks have reached stability according to the stability indicators. This shows one strength of the assessment method since it seems to cope with a number of framework dissimilarities.

6.4. Observations on studied cases

We describe four major differences between the BGW and MFC framework that are invariant with respect to the stability assessment method, and thereby do not seem to affect the method.

The MFC framework is a commercially available white-box object-oriented framework with a heterogeneous anonymous market, whereas the BGW framework is a proprietary framework. A proprietary-owned framework seems to be a more typical study subject since it represents the more common situation in the software industry, and because the framework developing organization has to deal with explicit customers and customer requirements rather than distributing the framework to a mass market.

Second, the BGW framework is a visual builder (black-box) framework, whereas the MFC framework is a white-box framework. Our experience is that black-box frameworks generally contain a larger number of classes and inheritance hierarchies than white-box frameworks. The rationale for this is that the normal extension mechanism in white-box frameworks is subclassing through inheritance, whereas black-box (visual) frameworks use parameterization. To support parameterization, black-box frameworks generally include multiple 'options'-hierarchies containing concrete subclasses that can be used in framework instantiations.

A third difference is the size, measured in number of classes, of the two frameworks. The original BGW framework (322 classes) was originally 4.5 times larger than the first MFC framework (72 classes). Version 4 of the BGW framework (598 classes) is still 2.9 times larger than the fifth version of the MFC framework (233).

Finally, the domains covered by the frameworks are quite different. The MFC framework implements graphical user interface (GUI) functionality. Compared to most domains, the GUI domain is quite stable in the behaviour it is supposed to provide. The BGW domain is considerably less stable and the developers of the framework have had to incorporate impressive amounts of requirement changes and additions during its lifetime.

7. ASSESSMENT

The method has been applied to two different frameworks, which makes it possible to better assess the method with respect to the three management issues noted earlier, i.e., framework deployment, change impact analysis and benchmarking. Also, it gives a richer discussion about the effort required to apply the method, the need for tool support, and the need to experience advantages and disadvantages of the method. In the following paragraphs, we discuss how the method performed with respect to the matters mentioned above.

The outcome of the stability assessment method gives an indication of how stable a framework version is. The relative- and normalized-extent-of-change metrics especially provide information as to how large an impact the requirements had on the framework structure. If historical trends for the current and other frameworks exist for the metrics, the metric values give a 'rough' indication of the maintenance effort needed. This is with reservation for requirements that have architectural impact and thus affect the whole framework. Metric values deviating from the values in the stability indicators are indications of evolution, which is not desirable and eventually has to be handled explicitly if time schedule and budgets allow. Thus, the stability assessment method is useful for change impact analysis.

The framework deployment aspect is not directly addressed by the method. However, if the method is applied more frequently during the iterative development of the framework version, the extent-of-change metrics can provide information with respect to whether the framework is becoming more or less stable during the iterations. The other metrics in the stability assessment method also indicate stability or not depending on whether the values are deviating from the ones listed in the stability indicators. The observed trends in the metric values give useful information for deciding whether to release the framework version or to iterate the development or maintenance once again.

Finally, the issue of benchmarking is supported by the method since it collects a number of metrics and thereby provides empirical data about the evolution of a framework. A historical set of data from previous assessments together with other information can make the prediction of the current framework's evolution more accurate.

The method as such is fairly simple to apply, but its application requires that the set of metrics used is clearly defined and consistently interpreted, both between framework versions and different frameworks in order to provide comparable data. Tool support is highly recommended since a huge amount of data has to be dealt with. The tool used in our study [17] had problems with the large BGW framework. Some of the problems were related to the number of file directories used in the framework, but not apparently to the number of classes.

An advantage of the method is the possibility of changing the underlying set of metrics. This is useful if the frameworks developed or used adhere to a specific coding standard or a specific technical domain, e.g., distributed systems, or any other organizational issue. A minor disadvantage of the method is the calculation of the aggregated metrics, especially the class level metrics changes, if no tool support is provided.

The stability assessment method has a modest cost to apply, but offers a number of benefits. The metric set used in the method provides a comprehensive view of the framework since it addresses five different framework stability aspects: the indicators. Another benefit is the possibility of changing the metric set used in the method. It is also possible, as in our case, to select a metric set which provides information about inheritance structure changes.

8. RELATED WORK

In this section we present work related to ours that addresses framework evolution and stability.

Mattsson and Bosch [21] present a method for identifying evolution-prone modules in a framework, which has been applied on the BGW framework. The method uses historical data related to the classes in the framework. Thus, the purpose of the method is to identify those parts of the framework that change most and it does not address how the whole framework evolves.

The work by Roberts and Johnson [4] presents a pattern language that describes typical steps for the evolution of an object-oriented framework. Common steps in the evolution are the development of a white-box framework (extensive use of the inheritance mechanism), the transition to a black-box framework (extensive use of composition), development of a library of pluggable objects, etc. These steps give a coarse-grained description of a framework's evolution with respect to its technical maturity. However, they do not explicitly address and describe where and how the framework evolves when it has reached a certain state.

Assessments of frameworks have been addressed by Bansiya [10] and Erni and Lewerentz [20]. Bansiya's assessment approach is similar to ours since we have used it as a basis, as described earlier in this paper. Differences were discussed in Section 6.3. Erni and Lewerentz describe a metrics-based approach that supports incremental development of a framework. By measuring using a set of metrics (which requires full source code access), an assessment of the design quality of the framework can be performed. If the metric data are within acceptable limits, the framework is considered to be good enough from a metrics perspective; otherwise another design iteration has to be done. The approach is intended to be used during the design iterations before a framework is released, and does not consider long-term evolution of a framework, a restriction not characteristic of our stability assessment method.

9. CONCLUSION

Object-oriented framework technology has become common technology in object-oriented software development. An object-oriented framework is a reusable asset that constitutes the basis for a number of applications in the same domain. As with all software, frameworks tend to evolve. Once the framework has been deployed, new versions of a framework cause high maintenance cost for the products built with the framework. This fact, in combination with the high costs of developing and evolving an application framework, make it important to have a knowledge of the nature of framework evolution.

Reusable frameworks have to be considered as in-house products, thus requiring long-term commitment for the maintenance and evolution of the framework. We have presented one method, stability assessment that assesses the stability of a framework version, which is useful in this organizational and long-term context. The method is metric-based and the set of metrics used is replaceable with other metric sets, which may be required due to specific domains and/or development processes.

Through applying the stability method on two different frameworks, both in the proprietary and commercial domain, we have formulated five experience-based framework stability indicators which can be used for indicating framework stability.

- Stable frameworks tend to have narrow and deeply inherited class hierarchy structures, characterized by high values for the average depth of inheritance (above 2.1) of classes and low values for the average width of inheritance hierarchies (below 0.85).
- A stable framework has an NSI/DSC (number of single inheritance classes divided by design size in classes) ratio just above 0.8 if multiple inheritance is seldom used in the framework. That is, the number of subclasses in a stable framework is just above 80%.
- The normalized ADCC (average of directly coupled classes) metric is going towards 1.0 or just below for stable frameworks.
- The normalized-extent-of-change metric is below 0.4 for a stable framework.
- A stable framework exhibits a relative-extent-of-change value of less than 25%.

Through the application of the method on two industrial frameworks, we have validated the method's applicability for assessing framework stability in an industrial context.

Currently, little knowledge about the stability and evolution of object-oriented frameworks is available. Using the stability assessment method's results can provide management with useful information. That information can help management make better informed decisions about an object-oriented framework's evolution, especially with respect to change impact analysis, benchmarking and, to some extent, framework deployment.

Nevertheless, we believe that the framework stability assessment method and the framework stability hypotheses discussed in this paper provide a useful and industrially applicable approach to assessing the stability of object-oriented frameworks. However, we believe that some qualitative analysis is necessary to complement the metrics-based stability assessment. As part of our future work, we intend to collect data from more object-oriented frameworks in order to validate further our framework stability indicators.

REFERENCES

1. Bosch J, Molin P, Mattsson M, Bengtsson P, Fayad ME. 1999. Framework problems and experiences. In *Object-Oriented Application Frameworks: Problems & Perspectives* Fayad ME, Schmidt DC, Johnson RE (eds.) John Wiley & Sons Inc.: New York NY; 55–82.
2. Mattsson M. 1996. Object-oriented frameworks – a survey of methodological issues. *Licentiate Thesis LU–CS–TR: 96–167*, Department of Computer Science, Lund University: Lund, Sweden.
3. Molin P, Ohlsson L. 1996. The points and deviations pattern language of fire alarm systems. In *Proceedings of the International Conference on Pattern Languages for Program Design 2*; Addison Wesley Longman Inc.: Reading MA; 431–445.
4. Roberts D, Johnson R. 1996. Patterns for evolving frameworks. In *Proceedings of the International Conference on Pattern Languages of Program Design 3*; Addison Wesley Longman Inc.: Reading MA; 471–486.
5. Moser S, Nierstrasz O. 1996. The effect of object-oriented frameworks on developer productivity. *IEEE Computer* **29**(9):45–51.
6. Geary D. 1997. *Graphic Java 1.1: Mastering the AWT*; Sun Microsystems Press: Mountain View CA.
7. Shepherd G, Wingo S. 1994. *MFC Internals: Inside the Microsoft Foundation Class Architecture*; Addison-Wesley Publishing Co.: Reading MA.
8. Weinand A, Gamma E, Marty R. 1989. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming* **10**(2):63–87.
9. Bosch J. 1999. Measurement systems framework. In *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, Fayad ME, Schmidt DC, Johnson RE (eds). John Wiley & Sons Inc.: New York NY; 117–206.
10. Bansiya J. 1999. Evaluating structural and functional stability. In *Object-Oriented Application Frameworks: Problems & Perspectives*, Fayad ME, Schmidt DC, Johnson RE (eds). John Wiley & Sons Inc.: New York NY; 599–616.

11. Mattsson M. 1999. Effort distribution in a six year industrial application framework project. In *Proceedings International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; 326–333.
12. Mattsson M, Bosch J. 1999. Characterizing stability in evolving frameworks. In *Proceedings 29th International Conference on Technology of Object-Oriented Languages and Systems*; IEEE Computer Society Press: Los Alamitos CA; 118–130.
13. Johnson R, Russo V. 1991. *Reusing object-oriented design*. *Technical Report UIUCDCS 91–1696*, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign IL.
14. Prieto-Diaz R, Arango G. 1991. Domain analysis and software systems modeling. IEEE Computer Society Press: Los Alamitos CA; 9–88.
15. Schäfer W, Prieto-Diaz R, Matsumoto M. 1994. *Software Reusability*; Ellis-Horwood Ltd.: New York NY; 17–49.
16. Gamma E, Helm R, Johnson R, Vlissides J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*; Addison-Wesley Publishing Co.: Reading MA.
17. Bansiya J. 1997. *A hierarchical model for quality assessment of object-oriented design*. Doctoral Dissertation, Computer Science Department, University of Alabama at Huntsville, Huntsville AL.
18. Chidamber SR, Kemerer CF. 1994. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering* **20**(6):476–493.
19. Li W, Henry S. 1993. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* **23**(2):111–122.
20. Erni K, Lewerentz C. 1996. Applying design metrics to object-oriented frameworks. In *Proceedings of the 1996 3rd International Metrics Symposium*; IEEE Computer Society Press: Los Alamitos CA; 64–74.
21. Mattsson M, Bosch J. 1999. Observations on the evolution of an industrial OO framework. In *Proceedings International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; 139–145.

AUTHORS' BIOGRAPHIES

Michael Mattsson is an Assistant Professor of Software Engineering at the University of Karlskrona/Ronneby, Sweden. He received an M.Sc. degree in Mathematics from Växj University, Sweden, and a Ph.D degree in Software Engineering from University of Karlskrona/Ronneby, Sweden. His research activities include object-oriented framework integration and evolution, and software product line architecture. E-mail: Michael.Mattsson@ipd.hk-r.se

Jan Bosch is a Professor of Software Engineering at the University of Karlskrona/Ronneby, Sweden, where he heads the RISE (Research In Software Engineering) research group. He received an M.Sc. degree from the University of Twente, The Netherlands, and a Ph.D. degree from Lund University, Sweden. His research activities include software architecture design, software product lines, object-oriented frameworks and component-oriented programming. Jan has organized many workshops and served on a varietey of programme committees. E-mail: Jan.Bosch@ipd.hk-r.se